Annotated multisemantics to prove Non-Interference analyses

Gurvan Cabon, Alan Schmitt

Annotated multisemantics to prove Non-Interference analyses

Gurvan Cabon
Inria
Rennes, France

Alan Schmitt
Inria
Rennes, France

ABSTRACT

The way information flows into programs can be difficult to track. As non-interference is a hyperproperty relating the results of several executions of a program, showing the correctness of an analysis is quite complex. We present a framework to simplify the certification of the correction proof of such analyses. The key is capturing the non-interference property through an annotated semantics based on the execution of the program and not simply its result. The approach is illustrated using a small While language.

CCS CONCEPTS

• **Security and privacy → Formal methods and theory of security**;

KEYWORDS

multisemantics, non-interference, pretty-big-step, annotation

1 INTRODUCTION

Non-interference can be defined as a program property that give guaranties on the independence of specific (public) outputs of a program from specific (secret) inputs. Non-interference is a hyperproperty [9]: it does not depend on one particular execution of the program (unlike illegal memory access for example), but on the results of several executions.

To develop a certified system verifying information flows, such as non-interference, we propose to only rely on the execution of the program, and thus investigate such properties using directly the derivation tree of an execution.

Considering a single execution is clearly not sufficient to determine if a program has the non-interference property. Surprisingly, studying every execution independently is also not sufficient. This is why we propose a formal approach that builds, from any semantics respecting a certain structure, a multisemantics that allows to reason on several executions simultaneously. Adding annotations to this multisemantics lets us capture the dependencies between inputs and outputs of a program.

We show that our approach is correct, i.e., annotations correctly capture non-interference. This allows analyses (systems detecting information leak giving non-interferent guarantees only when the tested program is actually non-interferent) to be proven correct as the dependencies are a simple property of the multisemantics defined by induction.

PLAS'17, October 30, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5099-0/17/10…$15.00
https://doi.org/10.1145/3139337.3139344

To demonstrate our approach, we present a small WHILE language and its semantics and build its annotated multisemantics.

Contributions. This paper provides a systematic transformation of a Pretty-Big-Step semantics into an annotated multisemantics that correctly captures dependencies as a property of the derived semantics. It does not provide an analysis, but a framework that can be used to formally prove analyses. The approach is partially formalized in the Coq proof assistant [12]: among the lemmas shown here, the lemmas of section 4.2 and of appendix B are proven with the Coq proof assistant.

Outline. In Section 2, we present the non-interference property and we give an intuition of our approach. In Section 3, we present the semantics format we use and show how a WHILE semantics is expressed in that format. In Sections 4 and 5, we describe how the multisemantics is systematically built and we extend it with annotations. In Section 6, we state and prove that the annotated multisemantics correctly capture non-interference. In Section 7, we compare our approach to previous works. We conclude in Section 8.

2 NON-INTERFERENCE

Suppose we have a programming language in which variables can be private or public, and where the programs can take variables as parameters. We say a program is non-interferent if, for any pair of execution that differs only on the private parameters, the values of the public variables are the same. In other words, changing the value of the private variables does not influence the public variables. Or in yet other words, the public variables do not depend on the private variables: there is no leak of private information.

Definition 2.1 (Termination-Insensitive Non-interference).
A program is *Non-interferent* if, for any pair of **terminating** executions starting with **different values in the private variables**, the executions end with the **same value in the public variables**.

In this work, we only consider finite program executions. We now illustrate through examples of increasing complexity where leaks of private information may happen and how one may detect them. As a simple first example, consider the naive program in Figure 1, where `public` is a public parameter and `secret` is a private variable. It is clearly interferent (or not non-interferent): changing the value of `secret` changes the value of `public`. This is a *direct flow* of information because the value of `secret` is directly assigned into `public`.

```
public := secret
```

Figure 1: Example of naive interference

Unfortunately, interference is not simply the transitive closure of direct flows. It may also come from the context in which a particular

instruction is executed. For example, Figure 2 shows a program with an *indirect flow*. The value of `secret` is not directly stored into `public` but the condition in the if statements ensures that in each case `secret` receives the value of `public`. One may thus detect interference by taking into account the context in which an assignment takes place. Any single execution of the program of Figure 2 would then witness the interference.

```
if secret
  then public := true
  else public := false
```

Figure 2: Example of indirect flow

Another source of interference is the fact that *not* executing a part of the code can provide information. This is often called *masking*. For example, Figure 3 shows such a program. In the case where `secret` is false, the variable `public` is not modified, so this execution does not witness the interference, even when taking the context into account. The other execution, where `secret` is true, does witness the interference. Hence a further refinement to detect interference would be to consider all possible executions of a program.

```
public := false
if secret
  then public := true
  else skip
```

Figure 3: Example of indirect flow with a mask

Unfortunately, this is not sufficient. In the example shown in Figure 4, we can see that there exists no single execution where the flow can be inferred. In the left execution, `public` depends on `y`, which is not modified by the execution. In the right execution, `public` still depends on `y`, which itself depends by indirect flow on `x`, which is not modified by the execution. Hence in both cases there seems to be no dependency on `secret`. Yet, we have `public` = `secret` at the end of both execution, so the secret is leaked. Looking at every execution *independently* is not enough.

To recover the inference of information flow as a property of an execution, we propose a different semantics where multiple executions are considered in lock-step, so that one may combine the information gathered by several executions. In the case of Figure 4, we can see that `x` depends on `secret` in the first execution at the end of the first `if`. Hence, in the second execution, `x` must also depend on `secret`, as the fact that not modifying it is an information flow. We can similarly deduce that `y` depends on `x` in both executions, hence `public` transitively depends on `secret`.

In some sense, we propose to internalize an approximation of the non-interference hyperproperty in a property of a refined semantics. Our approach gives the ability to reason inductively on the refined semantics and construct formal proofs of correctness of analyses.

3 PRETTY-BIG-STEP

As we aim to provide a generic framework independent of a specific programming language, we need a precise and simple way

```
x := true
y := true
if secret
  then x := false
  else skip
if x
  then y := false
  else skip
public := y
```

$secret = true$	$secret = false$
`x := true`	`x := true`
`y := true`	`y := true`
`if secret`	`if secret`
`  then x := false`	~~`  then x := false`~~
~~`  else skip`~~	`  else skip`
`if x`	`if x`
~~`  then y := false`~~	`  then y := false`
`  else skip`	~~`  else skip`~~
`public := y`	`public := y`
$public = true$	$public = false$

(executed code, ~~non-executed code~~)

Figure 4: Running Example

to describe its semantics. The Pretty-Big-Step semantics [8] is not only concise, it has been shown to scale to complex programming languages while still being amenable to formalization with a proof assistant [7]. We slightly modify the definition of Pretty-Big-Step to make it more uniform and to simplify the definition of non-interference.

3.1 Canonical structure

Memory model. We propose to model non-interference by making explicit the inputs of a program and its outputs. We do not consider interactive programs, so each input is a constant single value, for instance an argument of the program. Outputs, however, consists of lists of values, as we allow a program to send several values to a given output.

Formally, we consider given a set of values Val and a set of variables Var. We define the *memory* as a triplet (E_i, E_x, E_o), where $E_i \in Env_i$ represents the inputs of a program as a read-only mapping from each input to a value, $E_x \in Env_x$ represents run-time environment as a read-write mapping from each variable to a value, and $E_o \in Env_o$ represents the outputs of a program, as a write-only mapping of each output to a list of values, accumulated in the output. To simplify, we consider inputs and outputs to be indexed by an integer.

$$
\begin{aligned}
&Inputs := \mathbb{N} \\
&Outputs := \mathbb{N} \\
&Env_i := Inputs \mapsto Val \\
&Env_x := Var \mapsto Val \\
&Env_o := Outputs \mapsto Val\ list \\
&Mem := Env_i \times Env_x \times Env_o
\end{aligned}
$$

Semantics. The Pretty-Big-Step semantics is a constrained Big-Step semantics where each rule may only have 0, 1, or 2 inductive premises. In addition, one only needs to know the state and term under evaluation to decide which rule applies. To illustrate the Pretty-Big-Step approach, let us consider the evaluation of a conditional. It may look like this is Big-Step format.

$$\textsc{IfTrue}\ \frac{M, e \to (M', v) \qquad v = true \qquad M', s_1 \to M''}{M, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to M''}$$

Although this rule only has two inductive premises, one has to partially execute it to know it if is applicable (in this case if e evaluates to $true$). In Pretty-Big-Step, one first evaluates e, then passes control to another rule to decide which branch to evaluate. Additional constructs are needed to describe these intermediate steps, they are called *extended terms*, often written with a $_1$ or $_2$ subscript, and they need previously computed values. Here are the rules for evaluating a conditional in Pretty-Big-Step.

$$\textsc{If}\ \frac{M, e \to (M', v) \qquad (M', v), \texttt{If}_1\ s_1\ s_2 \to M''}{M, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to M''}$$

$$\textsc{IfTrue}\ \frac{M, s_1 \to M'}{(M, true), \texttt{If}_1\ s_1\ s_2 \to M'}$$

Formally, rules are in three groups shown in Figure 5: (i) axioms, the rules with no inductive premise; (ii) rules 1, the rules with one inductive premise; (iii) rules 2, the rules with two inductive premises.

Rules may either return a memory and a value, or just a memory. Conversely, in Pretty-Big-Step, rules may take as input a memory and zero, one, or several values. To account for this in a uniform way, we define a state σ as a pair of a memory and a list of values, called an *extra*. We write $extra(\sigma)$ to refer to the list of values in a state σ. The result of evaluating an expression is a state whose extra is a singleton list containing the resulting value. To simplify notations, we omit the extra when it is an empty list.

$Extra := List(Val)$
$State := Mem \times Extra$

A rule is entirely defined by the following components.

- Axioms
  - $t : term$, the term on which the axiom can be applied;
  - $ax : State \to State\ option$, a function that give the resulting state given the initial state.
- Rule 1
  - $t : term$, the term on which the rule 1 can be applied;
  - $up : State \to State\ option$, a function that returns the new state in which t_1 will be evaluated;
  - $t_1 : term$, a term to evaluate in order to continue the derivation.
- Rule 2
  - $t : term$, the term on which the rule 2 can be applied;
  - $up : State \to State\ option$, a function returning the state in which the term t_1 has to be applied;
  - $next : State * State \to State\ option$, a function giving the state in which t_2 had to be derived depending on the initial state and the result of the derivation of t_1;
  - $t_1, t_2 : term$, the terms to derive in order to get the result for t;
  - $prod\_extra$, a boolean value indicating if the evaluation of t_1 produces an extra.

The functions ax, up, and $next$ are functions returning a *State option* because these functions have no image for some states. For example, the rule IfTrue above is defined only when the state has a single extra that is the boolean value $true$. The $prod\_extra$ boolean is used to distinguish rules that produce an intermediate state with a non-empty extra to those who produce one with an empty extra. It is only used in Section 5 when annotating rules.

$$\text{Ax}\ \frac{}{\sigma, t \to ax(\sigma)} \qquad \text{R\_1}\ \frac{up(\sigma), t_1 \to \sigma'}{\sigma, t \to \sigma'}$$

$$\text{R\_2}\ \frac{up(\sigma), t_1 \to \sigma_1' \qquad next(\sigma, \sigma_1'), t_2 \to \sigma'}{\sigma, t \to \sigma'}$$

Figure 5: Types of rule for a Pretty-Big-Step semantics

For clarity reasons, Figure 5 assumes $ax(\sigma)$, $up(\sigma)$ and $next(\sigma, \sigma_1')$ return actual states and not an optional states. Rules are not defined when the results are None.

The intuition behind the rules Pretty-Big-Step is the following.

- If the evaluation is immediate, we can directly give the results (e.g., the evaluation of a skip statement or a constant). This behavior corresponds to an axiom.
- If the evaluation needs to branch depending on a previously computed value, stored as an extra, then a rule 1 is used. This is used for instance after evaluating the condition in a conditional statement.
- If the evaluation needs to first inductively compute an intermediate result, then a rule 2 is used. The intermediate result is used to compute the next state with which the evaluation continues.

We thus impose the following additional requirements. For rules 1 and rules 2, if up is defined, then it must not change nor inspect the memory, i.e., it can only change the extra part of the state, and this change is a function of the previous extra: $up(M, e) = Some(M', e') \implies M' = M \wedge e' = f(e)$. For rules 2, if $next$ is defined, then the new memory is the memory of the second argument, and the new extra only depends on the extras of the arguments: $next((M_1, e_1), (M_2, e_2)) = Some(M, e) \implies M = M_2 \wedge e = g(e_1, e_2)$. Finally, given a term and an extra, at most one rule applies.

3.2 WHILE language

To illustrate our approach, we introduce a small WHILE language. In this language, we distinguish two kinds of terms: expressions and statements. We first give the syntax of the language and then its semantics in Pretty-Big-Step form.

Syntax. An expression is either a constant value, a variable, an input, or the binary operation between two expressions. A statement is either a no-op operation skip, a sequence of two statements, a

conditional, a while loop, an assignment of an expression into a variable, or an assignment of an expression into an output.

⟨*expression*⟩ e ::= `Const n` | `Var x` | `Input n` | `Op e e`

⟨*statement*⟩ s ::= `Skip` | `Seq s s` | `If e s s` | `While e s` | `Assign x e` | `Output n e`

We add to the expressions and statements the extended terms required by the Pretty-Big-Step format.

⟨*expression*⟩ e ::= ... | `Op1 e` | `Op2`

⟨*statement*⟩ s ::= ... | `Seq1 s` | `If1 s s` | `While1 e s` | `While2 e s` | `Assign1 x` | `Output1 n`

Semantics. To simplify the reading of the rules and the examples, we use some usual notations.

c for `Const` c
x for `Var` x
e_1 `op` e_2 for `Op` e_1 e_2
$s_1; s_2$ for `Seq` s_1 s_2
$;_1\ s_2$ for `Seq1` s_2
$x := e$ for `Assign` x e
$x :=_1$ for `Assign1` x
`if` e `then` s_1 `else` s_2 for `If` e s_1 s_2
$\texttt{If}_1\ s_1\ s_2$ for `If1` s_1 s_2
`while` e `do` s for `While` e s
$\texttt{while}_1\ e$ `do` s for `While1` e s
$\texttt{while}_2\ e$ `do` s for `While2` e s
$f[x \mapsto v]$ denotes the function $y \mapsto \begin{cases} v & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$.

As an example of the Pretty-Big-Step semantics, consider the evaluation of a conditional. The evaluation of `if` b `then` s_1 `else` s_2 starts with the evaluation of the guarding condition b. The result is passed in an extra to the extended statement $\texttt{If}_1\ s_1\ s_2$. We then have two rules to evaluate $\texttt{If}_1\ s_1\ s_2$, one for each possible case for the extra.

4 MULTISEMANTICS

The first step of our approach is to derive a new semantics where several derivations are considered at once. We do not simply want a set of derivations, but a *multiderivation* where applications of the same rule at the same point in the derivation are shared.

We use the following notation to represent multiderivations

$$t \Downarrow \mu$$

where $\mu \subseteq \mathit{State} \times \mathit{State}$ is a relation between states. From now on, we refer to such a μ as a *multistate*. Intuitively, a multistate relates states that are before and after the execution of the term. Formally, for every pair $(\sigma, \sigma') \in \mu$, we should have

$$\sigma, t \to \sigma'$$

which is a property of the multisemantics that we state in Section 4.2 and have proven in Coq.

We need a few helper functions to define the multisemantics. First, for every function $f : X \mapsto Y\ \mathit{option}$, we define the relation $f_{Some}(S) \in X \times Y$ between any element of $S \subseteq X$ that has any image by f of the form *Some* y with $y \in Y$.

$$f_{Some}(S) = \begin{cases} \{(x,y) \mid x \in S \wedge f(x) = \mathit{Some}\ y\} & \text{if } \forall x \in S, f(x) = \mathit{Some}\ y \\ \text{undefined} & \text{otherwise} \end{cases}$$

Second, we define operators to extract the set of first and second components of a relation.

$$\texttt{fst}(r) = \{x \mid (x,y) \in r\}$$
$$\texttt{snd}(r) = \{y \mid (x,y) \in r\}$$

Third, we define the *strict* relation composition operator $\circ$, for every pair of relations r_1, r_2.

$$r_1 \circ r_2 = \begin{cases} \{(x,z) \mid \exists y, (x,y) \in r_1 \wedge (y,z) \in r_2\} & \text{if } \texttt{snd}(r_1) = \texttt{fst}(r_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

This operator is associative and propagates undefinedness, so we avoid using parentheses.

Finally, we define an operator on relations $\overrightarrow{\cdot}$ that takes a relation and returns a new relation where the left-hand side is remembered in the right-hand side.

$$\overrightarrow{r} = \left\{ (\sigma, (\sigma, \sigma_1')) \,\middle|\, (\sigma, \sigma_1') \in r \right\}$$

$$\textsc{MultiAx}\ \frac{\mu = ax_{Some}(\texttt{fst}(\mu)) \qquad \mu \neq \emptyset}{t \Downarrow \mu}$$

$$\textsc{MultiR1}\ \frac{t_1 \Downarrow \mu_1 \qquad \mu = up_{Some}(\texttt{fst}(\mu)) \circ \mu_1}{t \Downarrow \mu}$$

$$\textsc{MultiR2}\ \frac{t_1 \Downarrow \mu_1 \qquad t_2 \Downarrow \mu_2 \qquad \mu_n = \overrightarrow{up_{Some}(\texttt{fst}(\mu)) \circ \mu_1} \qquad \mu = \mu_n \circ next_{Some}(\texttt{snd}(\mu_n)) \circ \mu_2}{t \Downarrow \mu}$$

Figure 7: Translation of Pretty-Big-Step to multisemantics

4.1 Canonical structure

Figure 7 shows how to derive a rule in the multisemantics from a rule in Pretty-Big-Step style. There are three cases as there are three kinds of Pretty-Big-Step rules.

In order to derive an axiom, the multistate should be consistent with the ax function for every pair, that is for every pair (σ, σ') of the multistate, $ax(\sigma) = \mathit{Some}\ \sigma'$. We forbid μ to be empty because it would correspond to multiderivations that have no meaning. Deriving a rule 1 can be done if for every pair (σ, σ') in the multistate, there exists a state σ_1 such that $up(\sigma)$ is of the form *Some* σ_1 and (σ_1, σ') is a pair of a multistate obtained by derivation of t_1. To derive a rule 2, for every pair $(\sigma, \sigma') \in \mu$, their should exists three states $\sigma_1, \sigma_1', \sigma_2$ such that:

- $up(\sigma)$ is of the form *Some* σ_1
- (σ_1, σ_1') is a pair of a multistate obtained by derivation of t_1
- $next(\sigma, \sigma_1')$ is of the form *Some* σ_2
- (σ_2, σ') is a pair of a multistate obtained by derivation of t_2

$$\textsc{Cst}\ \frac{}{M, c \to (M, c)} \qquad \textsc{Var}\ \frac{E_x(x) = v}{(E_i, E_x, E_o), x \to ((E_i, E_x, E_o), v)}$$

$$\textsc{Op}\ \frac{M, e_1 \to (M'', v_1) \qquad (M', v_1), \mathsf{op}_1\ e_2 \to (M'', v)}{M, e_1\ \mathsf{op}\ e_2 \to (M'', v)} \qquad \textsc{Op1}\ \frac{M, e_2 \to (M', v_2) \qquad (M', (v_1, v_2)), \mathsf{op}_2 \to M'', v}{(M, v_1), \mathsf{op}_1\ e_2 \to M'', v}$$

$$\textsc{Op2}\ \frac{v = v_1\ op\ v_2}{(M, (v1, v2)), \mathsf{op}_2 \to (M, v)} \qquad \textsc{Input}\ \frac{E_i(n) = v}{(E_i, E_x, E_o), \mathtt{Input}\ n \to ((E_i, E_x, E_o), v)} \qquad \textsc{Skip}\ \frac{}{M, \mathtt{skip} \to M}$$

$$\textsc{Seq}\ \frac{M, s_1 \to M' \qquad M', ;_1\ s_2 \to M''}{M, s_1; s_2 \to M''} \qquad \textsc{Seq1}\ \frac{M, s \to M'}{M, ;_1\ s \to M'} \qquad \textsc{If}\ \frac{M, e \to (M', v) \qquad (M', v), \mathtt{If}_1\ s_1\ s_2 \to M''}{M, \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \to M''}$$

$$\textsc{IfTrue}\ \frac{M, s_1 \to M'}{(M, true), \mathtt{If}_1\ s_1\ s_2 \to M'} \qquad \textsc{IfFalse}\ \frac{M, s_2 \to M'}{(M, false), \mathtt{If}_1\ s_1\ s_2 \to M'}$$

$$\textsc{While}\ \frac{M, e \to (M', v) \qquad (M', v), \mathtt{while}_1\ e\ \mathtt{do}\ s \to M''}{M, \mathtt{while}\ e\ \mathtt{do}\ s \to M''} \qquad \textsc{WhileFalse}\ \frac{}{(M, false), \mathtt{while}_1\ e\ \mathtt{do}\ s \to M}$$

$$\textsc{WhileTrue1}\ \frac{M, s \to M' \qquad M', \mathtt{while}_2\ e\ \mathtt{do}\ s \to M''}{(M, true), \mathtt{while}_1\ e\ \mathtt{do}\ s \to M''} \qquad \textsc{WhileTrue2}\ \frac{M, \mathtt{while}\ e\ \mathtt{do}\ s \to M'}{M, \mathtt{while}_2\ e\ \mathtt{do}\ s \to M'}$$

$$\textsc{Asg}\ \frac{M, e \to (M', v) \qquad (M', v), x :=_1 \to M''}{M, x := e \to M''} \qquad \textsc{Asg1}\ \frac{E'_x = E_x[x \mapsto v]}{((E_i, E_x, E_o), v), x :=_1 \to (E_i, E'_x, E_o)}$$

$$\textsc{Output}\ \frac{M, e \to (M', v) \qquad (M', v), \mathtt{Ouput}_1\ n \to M''}{M, \mathtt{Ouput}\ n\ e \to M''} \qquad \textsc{Output1}\ \frac{E'_o = E_o[n \mapsto v :: E_o(n)]}{((E_i, E_x, E_o), v), \mathtt{Ouput}_1\ n \to (E_i, E_x, E'_o)}$$

Figure 6: Rules of the Pretty-Big-Step semantics

Because we need σ to determine $next(\sigma, \sigma'_1)$, we use the $\overrightarrow{\cdot}$ operator to remember σ.

These rules are not sufficient in the general case as they force every derivation to have the same structure. For example, when trying to derive an if statement in the multisemantics, all of the derivations have to go in the same branch. The multiderivation for a conditional has the following root.

$$\textsc{MultiIf}\ \frac{b \Downarrow \mu_1 \qquad \mathtt{If}_1\ s_1\ s_2 \Downarrow \mu_2 \qquad \mu_n = \overrightarrow{up_{Some}(\mathtt{fst}(\mu))} \circ \mu_1}{\mu = \mu_n \circ next_{Some}(\mathtt{snd}(\mu_n)) \circ \mu_2}{\mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \Downarrow \mu}$$

To derive $\mathtt{If}_1\ s_1\ s_2$ there are two options. Either

$$\textsc{MultiIfTrue}\ \frac{s_1 \Downarrow \mu_1 \qquad \mu = up_{Some}(\mathtt{fst}(\mu)) \circ \mu_1}{\mathtt{If}_1\ s_1\ s_2 \Downarrow \mu}$$

where $up_{Some}(\mathtt{fst}(\mu)) = \{((M, true), M) \mid (M, true) \in \mathtt{fst}(\mu)\}$ and $\mathtt{fst}(\mu)$ only contains states of the form $(M, true)$, or

$$\textsc{MultiIfFalse}\ \frac{s_2 \Downarrow \mu_2 \qquad \mu = up_{Some}(\mathtt{fst}(\mu)) \circ \mu_2}{\mathtt{If}_1\ s_1\ s_2 \Downarrow \mu}$$

where $up_{Some}(\mathtt{fst}(\mu)) = \{((M, false), M) \mid (M, false) \in \mu\}$ and $\mathtt{fst}(\mu)$ only contains states of the form $(M, false)$.

As those options are incompatible, it is impossible to have a multiderivation for a conditional where the guard is evaluated differently for some states. To fix this, we add a Merge rule. This rule simply states that if it is possible to derive a term with two multistates, then it is also possible to derive it from the union of them. In the case of an if statement t_{if}, one may thus use two subderivations, one for each status of the guard, and merge them together.

$$\textsc{Merge}\ \frac{t \Downarrow \mu_1 \qquad t \Downarrow \mu_2}{t \Downarrow \mu_1 \cup \mu_2}$$

We do not restrict the use of the Merge rule. In practice, we only use it when we need to apply different rules to a multistate.

4.2 Expected properties

We now prove properties that show that multiderivations correspond to multiple derivations. First, if $t \Downarrow \mu$ is derivable, then for every pair $(\sigma, \sigma') \in \mu$, $\sigma, t \to \sigma'$ is derivable. A proof by induction on the multiderivation is straightforward.

Lemma 4.1. $\forall t \mu.\ t \Downarrow \mu \implies \forall (\sigma, \sigma') \in \mu.\ \sigma, t \to \sigma'$

The converse implication is not true, however. Figure 8 shows an example of a program allowing Pretty-Big-Step derivations of

arbitrary size. For every $k \in \mathbb{N}$, a derivation starting with the value k in the first input needs to unroll k times the while loop. Each of these derivations are finite but considering all of them together would require an infinite multiderivation.

```
n := In 1
i := 0
While (i<n) do
   i := i + 1
```

Figure 8: Counter example to the reciprocal of lemma 4.1

Nonetheless, when taking a finite number of Pretty-Big-Step derivations, we are able to derive them all together in the multi-semantics. Using the fact that a finite set can be described as the union of singletons (one for each element of the set), we can prove this with the two lemmas 4.2 and 4.3. The first one states that if a term is derivable in Pretty-Big-Step then it is derivable in the multisemantics with the corresponding singleton relation. The second lemma states that if a term is derivable with two multistates then it is derivable with the union of them. Finite multistates are sufficient for our purpose since finding interference only requires two derivations (or equivalently: proving non-interference only requires to inspect every pair of derivations).

Lemma 4.2. $\forall t\sigma,\sigma'.\ \sigma, t \to \sigma' \implies t \Downarrow \{(\sigma,\sigma')\}$

Lemma 4.3. $\forall t\mu_1\mu_2.\ t \Downarrow \mu_1 \implies t \Downarrow \mu_2 \implies t \Downarrow \mu_1 \cup \mu_2$

The first lemma is proved by induction on the Pretty-Big-Step derivation and the second one is a direct use of the Merge rule. We have formally proved these three lemmas in Coq.

5 ANNOTATIONS

We now present how multiderivations may be annotated to track information flows.

5.1 Construction of the annotations

Our annotations track the inputs on which every variable and output depends in a dependency environment of type *Dep*, typically written *D*. Additionally, we track the *context dependency CD* of the current computation. It has type *CtxtDep*, a set of inputs, and it represents the dependency of the context in which the current expression or statement is evaluated. The context dependency is used to track indirect flows, and is similar to program counter levels, although more precise.

$Dep := (Var \cup Outputs) \mapsto Inputs\ set$
$CtxtDep := Inputs\ set$

An annotated derivation is written as follows.

$$CD, D, t \Downarrow \mu, D', VD'$$

$CD \in CtxtDep$ and $D \in Dep$ are the context dependency and the dependency environment before the execution. $D' \in Dep$ is the dependency environment after the execution of the term. $VD' \in CtxtDep$ is the set of inputs the computed value, i.e., the extra, depends on.

We suppose we are given, for each axioms, the inputs, variables, and outputs used by the rule. Formally, each axiom comes with four sets:

- $InputRead \subset Inputs$, the set of inputs the axiom may read;
- $VarRead \subset Var$, the set of variables the axiom may read;
- $VarWrite \subset Var$, the set of variables the axiom may write;
- $OutputWrite \subset Outputs$, the set of outputs the axiom may write.

These sets respect the following properties.

(1) If two states have identical extras, and their memories are equal on the inputs and variables that can be read by the axiom, then for every variable $x \in VarWrite$, the value stored in x after the axiom is the same in both memories.
(2) The value in variables not in $VarWrite$ are not modified by the axiom.
(3) If two states have identical extras, and their memories are equal on the inputs and variables that can be read by the axiom, then for every output $o \in OutWrite$, the value added to o after the axiom is the same in both memories.
(4) The value in outputs not in $OutWrite$ are not modified by the axiom.

More formally :

(1) $\forall\sigma_1\sigma_2.\quad (extra(\sigma_1) = extra(\sigma_2))$
$\quad\wedge\quad (\forall i \in InputRead.\ \sigma_1(i) = \sigma_2(i))$
$\quad\wedge\quad (\forall y \in VarRead.\ \sigma_1(y) = \sigma_2(y))$
$\quad\implies\quad (\forall x \in VarWrite.\ ax(\sigma_1)(x) = ax(\sigma_2)(x))$
(2) $\forall\sigma,\sigma'.\ \forall x \notin VarWrite.$
$ax(\sigma) = Some\ \sigma' \implies \sigma(x) = \sigma'(x)$
(3) $\forall\sigma_1\sigma_2.\quad (extra(\sigma_1) = extra(\sigma_2))$
$\quad\wedge\quad (\forall i \in InputRead, \sigma_1(i) = \sigma_2(i))$
$\quad\wedge\quad (\forall y \in VarRead, \sigma_1(y) = \sigma_2(y))$
$\quad\implies\quad \left(\begin{array}{l}\forall v_1 v_2.\ \forall o \in OutWrite.\\ \qquad ax(\sigma_1)(o) = v_1 :: \sigma_1(o)\\ \wedge\quad ax(\sigma_2)(o) = v_2 :: \sigma_2(o)\\ \implies v_1 = v_2\end{array}\right)$
(4) $\forall\sigma,\sigma'.\ \forall o \notin OutputWrite.$
$ax(\sigma) = Some\ \sigma' \implies \sigma(o) = \sigma'(o)$

The annotated semantics rules in Figure 9 are the multisemantics rules extended with annotation information.

The most complex case is the one for axioms. For every variable written by the axiom, we replace the dependency for that variable by the union of the current context dependencies, the inputs the axiom may read, and the dependencies of the variables the axiom may read. Note that this is a strong update: we throw away prior dependencies for that variable as it is overwritten. In contrast, for every output written by the axiom, we add the union of the current context dependencies, the inputs the axiom may read, and the dependencies of the variables the axiom may read to the old dependencies of the output. This is because the output is added to the list of previous outputs.

Rules 1 are simple to annotate: they propagate annotations.

The annotations for a Rule 2 depend on whether the first premise produces an extra. If it does not, no context dependency is added in the evaluation of the continuation (dependencies of side effects of the first premise are already recorded in D_1). If the rule produces an extra, then the dependencies of that extra VD_1 are added to the context dependencies to evaluate the continuation.

$$\textsc{aMultiAx}\ \frac{\mu = ax_{Some}(\texttt{fst}(\mu)) \qquad \mu \neq \emptyset}{CD, D, t \Downarrow \mu, D', VD'}$$

where

$$VD' = CD \cup InputRead \bigcup_{x \in VarRead} D(x)$$

$$\forall x.\ D'(x) = \begin{cases} VD' & \text{if } x \in VarWrite \\ D(x) & \text{otherwise} \end{cases}$$

$$\forall o.\ D'(o) = \begin{cases} VD' \cup D(o) & \text{if } o \in OutputWrite \\ D(o) & \text{otherwise} \end{cases}$$

$$\textsc{aMultiR1}\ \frac{CD, D, t_1 \Downarrow \mu_1, D_1, VD_1 \qquad \mu = up_{Some}(\texttt{fst}(\mu)) \circ \mu_1}{CD, D, t \Downarrow \mu, D_1, VD_1}$$

$$\textsc{aMultiR2}\ \frac{\begin{array}{c} CD, D, t_1 \Downarrow \mu_1, D_1, VD_1 \qquad CD', D_1, t_2 \Downarrow \mu_2, D_2, VD_2 \\ \mu_n = \overrightarrow{up_{Some}}(\texttt{fst}(\mu)) \circ \mu_1 \\ \mu = \mu_n \circ next_{Some}(\texttt{snd}(\mu_n)) \circ \mu_2 \end{array}}{CD, D, t \Downarrow \mu, D_2, VD_2}$$

where $CD' = CD \cup \begin{cases} VD_1 & \text{if } prod\_extra \\ \emptyset & \text{otherwise} \end{cases}$

$$\textsc{aMerge}\ \frac{CD, D, t \Downarrow \mu_1, D_1, VD_1 \qquad CD, D, t \Downarrow \mu_2, D_2, VD_2}{CD, D, t \Downarrow \mu_1 \cup \mu_2, D', VD_1 \cup VD_2}$$

where $D'(xo) = D_1(xo) \cup D_2(xo)$for all variable and output xo

Figure 9: Types of rule for an annotated multisemantics

An example of the first case is the sequence rule.

$$\textsc{aMultiSeq}\ \frac{\begin{array}{c} CD, D, s_1 \Downarrow \mu_1, D_1, VD_1 \qquad \mathbf{CD}, D_1, ;_1\ s_2 \Downarrow \mu_2, D_2, VD_2 \\ \mu_n = \overrightarrow{up_{Some}}(\texttt{fst}(\mu)) \circ \mu_1 \\ \mu = \mu_n \circ next_{Some}(\texttt{snd}(\mu_n)) \circ \mu_2 \end{array}}{CD, D, s_1; s_2 \Downarrow \mu, D_2, VD_2}$$

An example of the second case, where an extra is produced, is the rule for conditionals.

$$\textsc{aMultiIf}\ \frac{\begin{array}{c} CD, D, b \Downarrow \mu_1, D_1, VD_1 \\ \mathbf{CD} \cup \mathbf{VD\_1}, D_1, \texttt{If}_1\ s_1\ s_2 \Downarrow \mu_2, D_2, VD_2 \\ \mu_n = \overrightarrow{up_{Some}}(\texttt{fst}(\mu)) \circ \mu_1 \\ \mu = \mu_n \circ next_{Some}(\texttt{snd}(\mu_n)) \circ \mu_2 \end{array}}{CD, D, \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \mu, D_2, VD_2}$$

Finally the Merge rule simply merges the dependencies together by doing the pointwise union of the dependencies environments, and the union of the value dependencies. For instance, for a conditional where both branches are executed, the dependencies are the union of the dependencies of each branch.

```
if Input 1
    then Output 1 0
    else Output 1 0
```

Figure 10: A program for which the annotations will over-approximate

Note that annotations only approximates the notion of non-interference and we may capture dependencies that do not lead to interferences. For example, Figure 10 shows a program for which any annotated multiderivation will calculate that the output 1 depends on the input 1, although changing input 1 would not change the result. The loss of precision comes from the fact that we only track dependencies, and not the actual values being computed.

5.2 Capturing masking

In Figure 11, we have re-written the running example of Figure 4 in the WHILE language with the value of secret stored in the first input and the public variable being the first output. We now show how our approach captures the dependency.

```
x := true;
y := true;
if Input 1
    then x := false
    else skip;
if x
    then y := false
    else skip;
Output 1 y
```

Figure 11: The running example in the WHILE language

Consider two states, one with *false* in the first input and one with *true*. We derive the running example in the annotated multi-semantics. We write $D_\emptyset$ the empty dependencies environment, a function returning an empty set for every variable and output. We suppose x and y are already set to *true* and the dependencies are empty.

When evaluating the first `if` statement, we have to derive the condition `Input 1` and then derive each branch with a smaller relation (after applying rule Merge) depending on the condition. The reader can easily verify that the first branch is derived as

$$\frac{\cdots}{\{1\}, D_\emptyset, \texttt{If}_1\ x := false\ \texttt{skip} \Downarrow \mu_{true}, D_\emptyset[x \mapsto \{1\}], \{1\}}\ \textsc{aMltIfTrue}$$

and second branch is derived as

$$\frac{\cdots}{\{1\}, D_\emptyset, \texttt{If}_1\ x := false\ \texttt{skip} \Downarrow \mu_{false}, D_\emptyset, \{1\}}\ \textsc{aMltIfFalse}$$

where μ_{true} and μ_{false} are the singleton multistates relating only the corresponding states in the Pretty-Big-Step semantics for both derivation.

It leads us to derive the statement $\texttt{If}_1\ x := false\ \texttt{skip}$ with a merge rule as follow

$$\frac{\{1\}, D_\emptyset, \texttt{If}_1\ x := false\ \texttt{skip} \Downarrow \mu_{true}, D_\emptyset[x \mapsto \{1\}], \{1\} \quad \{1\}, D_\emptyset, \texttt{If}_1\ x := false\ \texttt{skip} \Downarrow \mu_{false}, D_\emptyset, \{1\}}{\{1\}, D_\emptyset, \texttt{If}_1\ x := false\ \texttt{skip} \Downarrow \mu, D_\emptyset[x \mapsto \{1\}], \{1\}}\ \textsc{aMerge}$$

where $\mu = \mu_{true} \cup \mu_{false}$.

Putting these together, the derivation of $P_1 = \texttt{if Input 1 then}\ x := false\ \texttt{else skip}$ is

$$\frac{\emptyset, D_\emptyset, \texttt{Input 1} \Downarrow \mu_{input}, D_\emptyset, \{1\} \quad \{1\}, D_\emptyset, \texttt{If}_1\ x := false\ \texttt{skip} \Downarrow \mu, D_\emptyset[x \mapsto \{1\}], \{1\}}{\emptyset, D_\emptyset, P_1 \Downarrow \mu_{P_1}, D_\emptyset[x \mapsto \{1\}], \{1\}}\ \textsc{aMltIf}$$

where μ_{input} and μ_{P_1} are the multistates each one relating two pairs of states corresponding the derivations in the Pretty-Big-Step semantics for the terms.

Without even going further, we already know that x depends on the first input. The second `if` statement has the same behavior: at the end we also infer that y depends on the first input.

Finally, when observing y, the dependency flows into the first output. If we call our program *runningExample* we have

$$\frac{\cdots}{\emptyset, D_\emptyset, runningExample \Downarrow \mu_{RE}, D, \{1\}}$$

where $D = D_\emptyset[x \mapsto \{1\}][y \mapsto \{1\}][1 \mapsto \{1\}]$ and μ_{RE} is the relation relating the two pairs of states appearing in the corresponding Pretty-Big-Step derivations. We can observe that we have $1 \in D(1) = \{1\}$.

5.3 Precision

As our framework relies on executions, we can potentially be more precise than static analyses. This is not surprising as we do not provide analyses, but a way to prove their correction. Thus, very precise analyses that can infer which branch of a conditional is taken can still be proven correct with our framework.

To illustrate this, we suppose that our language has been extended with the infix operators <=, == and +, which are respectively the lower or equal operator, the equal operator and the addition operator. We also introduce the logical *not* operator and we use a shortcut *isprime* to represent an expression returning *true* if i is prime and false otherwise (for the purpose of this example, it could just be a disjunction of equalities between i and all of the prime numbers smaller than 200). In the example of Figure 12, in any annotated multiderivation, the annotations will show that x does not depend depends on input 1 because in every execution, the loop will end up overwriting the value of x by the constant 0. It implies that output 1 depends on nothing. In the other hand, a syntactic method (for example we could adapt one from Sabelfeld and Myers approach [19]) approximates the dependencies after the `if` statement saying that x depends on input 1, and then the output 1 also depends on input 1.

Let t the program of Figure 12. We have the following result.

Lemma 5.1. *For every μ, if $\emptyset, D_\emptyset, t \Downarrow \mu, D', VD'$, then $D'(1) = \emptyset$.*

```
i := 0;
while i <= 100 or not(isprime(i)) do
  if i == 101
    then x:=0
    else x:=Input 1;
  i := i + 1;
Output 1 x
```

Figure 12: An example where the annotations do not over-approximate

Proof. First, we show that for any subderivation $CD, D, t' \Downarrow \mu, D', VD'$ where t' is the while loop, we have $D'(x) = \emptyset$. Then we deduce that $D'(1) = \emptyset$

We proceed by induction on $n = 101 - i$. It is possible because all states share the same value in the i variable (and μ is not empty).

The base case is when $i = 101$. In that case, rule IfTrue applies, and in the resulting dependency we have $D'(x) = \emptyset$ since at that point $CD = \emptyset$. More precisely, we evaluate `true`, which returns a $VD = \{\}$, then we do the assignment to x, which does a strong update of $D'(x)$ as $x \in VarWrite$ for rule Asg1.

For the inductive case $n > 0$, we have $i < 101$, then WhileTrue applies. First we derive the body of the while loop and then we derive the while loop with the value $i + 1$ in the variable i. The resulting dependency is then the one from the while loop with in the case $n - 1$, i.e. $D'(x) = \emptyset$.

When $i = 102$, rule WhileFalse applies, followed by rules Ouput and Output1, where $D'(1)$ is set to the union of $D(1) = \emptyset$ and $D(x)$. Hence $D'(1) = \emptyset$. □

6 CORRECTNESS

We now formally prove that the framework is correct.

6.1 Correctness theorem

We define $\Delta(\sigma_1, \sigma_2)$ as the set of variables and outputs on which the two states differ.

Definition 6.1. Let σ_1 and σ_2 be two states.

$$\Delta(\sigma_1, \sigma_2) = \{x \in Var \mid \sigma_1(x) \neq \sigma_2(x)\} \cup \{o \in Outputs \mid \sigma_1(o) \neq \sigma_2(o)\}$$

Two derivations are said to be *(I,o)-interferent* if a difference in only the inputs $I \subset Inputs$ results in a difference in the output $o \in Outputs$.

Definition 6.2. Let t be a term, I a finite set of private inputs and o a public output. t is *(I,o)-interferent* if there exist four states σ_1, σ_1', σ_2 and σ_2' such that $\sigma_1, t \to \sigma_1'$, $\sigma_2, t \to \sigma_2'$ and:

$\forall i' \in Inputs \backslash I, \sigma_1(i') = \sigma_2(i')$
$\wedge \quad \forall i \in I, \sigma_1(i) \neq \sigma_2(i)$
$\wedge \quad \Delta(\sigma_1, \sigma_2) = \emptyset$
$\wedge \quad extra(\sigma_1) = extra(\sigma_2)$
$\wedge \quad \sigma_1'(o) \neq \sigma_2'(o)$

We will note this formula $interf_{(I,o)}(\sigma_1, \sigma_1', \sigma_2, \sigma_2')$.

A term is then interferent if and only if there exists a finite set of inputs I and an output o such that it is *(I,o)-interferent.*

The fundamental theorem 6.3 is the main theorem we want to prove. It says that if we have two Pretty-Big-Step derivations showing that output o depends on inputs I, then their exists an annotated multiderivation with empty dependencies on the left such that the annotation shows this interference. By contraposition it means that if for every multiderivation we cannot show interference by the annotations, then the program is non-interferent.

Theorem 6.3 (Fundamental Theorem). $\forall t, \sigma_1, \sigma_1', \sigma_2, \sigma_2', I, o.$

$$
\begin{array}{lll}
 & \sigma_1, t \to \sigma_1' & \\
\wedge & \sigma_2, t \to \sigma_2' & \\
\wedge & interf_{(I,o)}(\sigma_1, \sigma_1', \sigma_2, \sigma_2') & \\
\Longrightarrow & \exists \mu, D', VD' \text{ such that} & \\
 & \quad \emptyset, D_\emptyset, t \Downarrow \mu, D', VD' & \\
 & \wedge \quad (\sigma_1, \sigma_1') \in \mu & \\
 & \wedge \quad (\sigma_2, \sigma_2') \in \mu & \\
 & \wedge \quad I \subseteq D'(o) &
\end{array}
$$

The fundamental theorem is a particular case of the more general lemma A.1 when we take $CD = \emptyset$ and $D = D_\emptyset$.

6.2 Proving an analysis

Given a program, proving the absence of information leakage with this framework would require considering every annotated multiderivation with exactly two pairs of states in the multistate and prove that there is no unwanted dependency. But proving interference requires only one annotated multiderivation. This allows us to use the framework to prove analyses.

Let us consider an analysis A. It is a function returning $true$ for at least each interferent program and may have some false-positives. But if the function returns $false$, it means the analyzed program satisfies the property of non-interference.

The standard way to prove the analysis A is the following :

Lemma 6.4.
$\forall P,$
if P *is interferent*
then $A(P)$

Such proofs are difficult to do by induction of the program since non-interference is an hyperproperty that is not defined by induction. When assuming the hypothesis "P is interferent", we only have information on what happens before two executions (the states differ only on some private inputs) and after (the resulting states differ on a public output). No information is given on what happens in the program.

Instead, if one uses our framework, he has to prove:

Lemma 6.5.
$\forall P, I, o$
if $CD, D, P \Downarrow \mu, D', VD' \wedge I \subseteq D'(o) \wedge I$ are private $\wedge$ o is public
then $A(P)$

because if P is interferent then the hypothesis of lemma 6.5 is satisfied for some I and o. This proof can be done by induction on the annotated multiderivations. It is easier to manipulate because assuming that we have a leaking annotated derivation gives us a whole derivation tree with annotations at each semantic step.

The drawback of our approach is that one cannot prove the correctness of an analysis that is more complete than our method.

7 RELATED WORK

Studies about non-interference take their roots in 1977 with E. Cohen [10] and D. E. Denning & P. J. Denning [11]; and then formalized in 1982 by J. A. Goguen & J. Meseguer [14] as following:

> One groups users, using a certain set of commands, is noninterfering with another group of users if what the first group does with those commands has no effect on what the second group of users can see.

There are several modern definitions of non-interference. In particular, non-interference may take into account the termination of an execution of the program. We thus have *termination-insensitive non-interference* [1], *termination-aware non-interference* [6], and *timing- and termination-sensitive non-interference* [16]. Our work considers termination-insensitive non-interference. To be able to deal with non-terminating executions, we would need to consider a coinductive version of the semantics.

A major inspiration of our work is the 2003 paper by A. Sabelfeld & A. C. Myers [19]. They give an overview of the information-flow techniques and show the many sources of potential interference. Our long-term goal is to evaluate our approach with the full Pretty-Big-Step semantics of JavaScript [7] and to show that [19] listed every possible source of information leak.

The thesis of G. Le Guernic [17] proposes and proves a precise dynamic analysis for non-interference. T. Austin and C. Flanagan also propose sound dynamic analyses for non-interference based on the *no-sensitive-upgrade* policy [2] and the *permissive upgrade* policy [3]. Our approach is similar in the sense that it is based on actual executions, but we consider every execution whereas these works monitor a single execution, modifying it if it is interferent. We believe, and should prove, that we are at least as precise as these works. Our goals are also quite different: they provide a monitor, we provide a framework to simplify the certification of analyses.

A. Sabelfeld and A. Russo [18, 20] prove several properties comparing static and dynamic approaches of non-interference. In particular, purely dynamic monitors can not be sound and permissive but it is possible for an hybrid monitor. Our framework could be a way to certify the correctness of such hybrid monitors.

G. Barthe, P.R. D'Argenio & T. Rezk [5] reduce the problem of non-interference of a program into a safety property of a transformation of the program. It allows to use standard techniques based on program logic for information flow verification. Our work is similar in the sense that we both transform a hyperproperty into a property. Self-composition achieves it by transforming the program, whereas we achieve it by extending the semantics in a mechanical way. In addition, our approach never inspects the values produced by the program, but only how it manipulates them. This is the reason why our approach is incomplete. For instance, we do not identify when two branches of a conditional do the same thing and we may flag it as interferent.

S. Hunt & D. Sands [15] present a family of semantically sound type system for non-interference. The main relation between the paper is the use of dependencies: a mapping from a variable to sets of variables they depend on in [15], a mapping from variables and outputs to set of inputs in our case. Our work is more precise as it does not use program points but actual executions. We also never consider the dependencies from branches of conditionals that are

taken by no execution, as illustrated in Figure 12. Finally, we do not propose an analysis, but a generic way to mechanically build the refined semantics.

D. Devriese and F. Piessens [13] introduce the notion of *secure multi-execution* allowing a sound and precise technique for information flow verification by execution a program multiple times with different security levels. Inspired by this work, T. Austin and C. Flanagan [4] present a new dynamic analysis for information flow based on *faceted values*. Our approach lies between secure multi-execution and faceted execution: we do not tag data but spawn multiple executions. In our pretty-big-step setting, however, the continuations of those executions are shared, in a way reminiscent of faceted execution.

8 CONCLUSION

In this paper, we presented a framework to automatically refine a semantics written in Pretty-Big-Step form into a new multisemantics able to consider *many derivations at once* and proved with the Coq proof assistant a correctness relation between the new and old semantics. We then presented an extension of the multisemantics with annotations that soundly approximates the notion of non-interference. The correctness proofs of the annotations is done by hand in the appendices. The final annotated multisemantics is a tool to prove the correctness of non-interference analyses.

Our next step is the full proof in Coq of the approach, followed by the extensions of the example language to show we can capture information flows in presence of functions and exceptions. We then want to apply the approach to certify existing analyses. Finally, we plan to refine the annotations in the Merge rule to inspect the results of computation, only adding dependencies when the results differ. We conjecture this will result in a framework that is complete in relation to non-interference.

REFERENCES

[1] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*. 333–348.

[2] Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/1554339.1554353

[3] Thomas H. Austin and Cormac Flanagan. 2010. Permissive Dynamic Information Flow Analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '10)*. ACM, New York, NY, USA, 3:1–3:12. https://doi.org/10.1145/1814217.1814220

[4] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 165–178.

[5] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011), 1207–1252.

[6] Nataliia Bielova and Tamara Rezk. 2016. A Taxonomy of Information Flow Monitors. In *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 46–67.

[7] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. San Diego, CA, USA, 87–100.

[8] Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 41–60.

[9] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[10] Ellis Cohen. 1977. Information Transmission in Computational Systems. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles (SOSP '77)*. ACM, New York, NY, USA, 133–139.

[11] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513.

[12] The Coq development team. 2016. *The Coq proof assistant reference manual*. http://coq.inria.fr Version 8.6.

[13] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. 109–124.

[14] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. 11–20.

[15] Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 79–90.

[16] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. 2011. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. 413–428.

[17] Gurvan Le Guernic. 2007. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. Ph.D. Dissertation. Kansas State University. http://tel.archives-ouvertes.fr/tel-00198621/fr/

[18] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. 186–199.

[19] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[20] Andrei Sabelfeld and Alejandro Russo. 2009. From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*. 352–365.

Appendices

Before starting the proofs, we recall important hypothesis over the Pretty-Big-Step semantics:

- For a given term and a given extra, there is a most one rule derivable,
- The functions *up* don't change the memory but only the extra.
- The functions *next* keep the memory of the second argument and the extra depends only on the extras of the arguments.

A PROOF OF CORRECTNESS OF THE ANNOTATIONS

Lemma A.1. $\forall t, \sigma_1, \sigma_1', \sigma_2, \sigma_2', I,$
$\sigma_1, t \to \sigma_1'$
$\implies \sigma_2, t \to \sigma_2'$
$\implies \forall i' \in Inputs \backslash I, \sigma_1(i') = \sigma_2(i')$
$\implies \forall i \in I, \sigma_1(i) \neq \sigma_2(i)$
$\implies \forall D, CD,$
$\forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x)$
$\implies (extra(\sigma_1) \neq extra(\sigma_2) \implies I \subset CD)$
$\implies \exists D', VD' \text{ such that}$
1. $CD, D, t \Downarrow \{(\sigma_1, \sigma_1'), (\sigma_2, \sigma_2')\}, D', VD'$
2. $\forall y \in \Delta(\sigma_1', \sigma_2'), I \subset D'(y)$
3. $extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD'$

Proof. Let $t, \sigma_1, \sigma_2, \sigma_1', \sigma_2', I$.

Let $\sigma_1, t \to \sigma_1'$ and $\sigma_2, t \to \sigma_2'$ be two Pretty-Big-Step derivations. Let's continue the proof by induction on the first derivation and then by a case matching on the second one.

First case. In the case of two different rules R and R', we necessarily have $extra(\sigma_1) \neq extra(\sigma_2)$ by hypothesis on the Pretty-Big-Step semantics. We will have to use the Merge rule.

Let suppose
$\forall i' \in Inputs \backslash I, \sigma_1(i') = \sigma_2(i')$
$\forall i \in I, \sigma_1(i) \neq \sigma_2(i)$
Let's have D and CD such that
$\forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x)$
$(extra(\sigma_1) \neq extra(\sigma_2) \implies I \subset CD)$
By this last hypothesis we have $I \subset CD$. Thanks to lemma B.1 there exists D_1', VD_1', D_2', VD_2' such that $CD, D, t \Downarrow \{(\sigma_1, \sigma_1')\}, D_1', VD_1'$ and $CD, D, t \Downarrow \{(\sigma_2, \sigma_2')\}, D_2', VD_2'$.
We can construct:

- $VD' = VD_1' \cup VD_2'$
- $D'(xo) = D_1'(xo) \cup D_2'(xo)$

We now have our 3 points:

(1) $CD, D, t \Downarrow \{(\sigma_1, \sigma_1'); (\sigma_2, \sigma_2')\}, D', VD'$ thanks to the merge rule
(2) $\forall y \in \Delta(\sigma_1', \sigma_2')$,
either, $y \in \Delta(\sigma_1, \sigma_2)$ and then by hypothesis, $I \subset D(y)$. Lemma B.2 ensures that $I \subset D'(y)$.
or, $\sigma_1(y) = \sigma_2(y)$ and we can assume (for symmetric reasons) that $\sigma_1(y) \neq \sigma_1'(y)$. Lemma B.3 ensures that $CD \subset D'(y)$ and thus $I \subset CD \subset D'(y)$
Anyway, $I \subset D'(y)$.
(3) $extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD_1'$ is a direct consequence of Lemma B.2

Second case. In the other case, both derivation is made by the same rule:

- Axiom:
$$Ax \frac{}{\sigma_1, t \to \sigma_1'}$$
and,
$$Ax \frac{}{\sigma_2, t \to \sigma_2'}$$
Let suppose
$\forall i' \in Inputs \backslash I, \sigma_1(i') = \sigma_2(i')$
$\forall i \in I, \sigma_1(i) \neq \sigma_2(i)$
Let's have D and CD such that
$\forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x)$
$(extra(\sigma_1) \neq extra(\sigma_2) \implies I \subset CD)$
We can construct:
– $VD' = CD \cup InputRead \bigcup_{x \in VarRead} D(x)$
–
$$D'(xo) = \begin{cases} VD' & \text{if } xo \in VarWrite \\ VD' \cup D(xo) & \text{if } xo \in OutputWrite \\ D(xo) & \text{otherwise} \end{cases}$$
We now have 3 points to prove:
(1) by construction of D' and VD', we have the derivation $D, CD, t \Downarrow \{(\sigma_1, \sigma_1'); (\sigma_2, \sigma_2')\}, D', VD'$
(2) $\forall y \in \Delta(\sigma_1', \sigma_2')$, we have 4 cases:
y is a variable and $y \in VarWrite$.
By hypothesis on the elements of $VarWrite$, either $(extra(\sigma_1) \neq extra(\sigma_2))$, $(\exists i' \in InputRead, \sigma_1(i') \neq \sigma_2(i'))$, or $(\exists x \in VarRead, \sigma_1(x) \neq \sigma_2(x))$. We can simplify this into: either $(extra(\sigma_1) \neq extra(\sigma_2))$, $(I \subset InputRead)$, or $(\exists x \in VarRead, x \in \Delta(\sigma_1, \sigma_2))$. It means that either $I \subset CD$, $I \subset InputRead$, or $(\exists x \in VarRead, I \subset D(x))$.
In the three cases, $I \subset VD'$, and thus $I \subset D'(y)$.
y is a variable and $y \notin VarWrite$.
By hypothesis on the elements not member of $VarWrite$, $\sigma_1(y) \neq \sigma_2(y)$ i.e. $y \in \Delta(\sigma_1, \sigma_2)$ and by hypothesis, $I \subset D(y) = D'(y)$.
y is an output and $y \in OutputWrite$.
Let's define l_1 and l_2 such that $\sigma_1'(y) = l_1 :: \sigma_1(y)$ and $\sigma_2'(y) = l_2 :: \sigma_2(y)$.
Since $l_1 :: \sigma_1(y) \neq l_2 :: \sigma_2(y)$, either $l_1 \neq l_2$, or $y \in \Delta(\sigma_1, \sigma_2)$.
With the same reasoning than for the first case: either $I \subset CD$, $(I \subset InputRead)$, $(\exists x \in VarRead, I \subset D(x))$, or $I \subset D(y)$. We have in every cases: $I \subset D'(y)$.
y is an output and $y \notin OutputWrite$.
By hypothesis on the elements not member of $OutputWrite$, $\sigma_1(y) \neq \sigma_2(y)$ i.e. $y \in \Delta(\sigma_1, \sigma_2)$ and by hypothesis, $I \subset D(y) = D'(y)$.
(3) if $extra(\sigma_1) \neq extra(\sigma_2)$ then by hypothesis $I \subset CD \subset VD'$

- R1:
$$R_1 \frac{up(\sigma_1), t_1 \to \sigma_1'}{\sigma_1, t \to \sigma_1'}$$

and,

$$R_1 \frac{up(\sigma_2), t_1 \to \sigma_2'}{\sigma_2, t \to \sigma_2'}$$

Let suppose
$\forall i' \in Inputs \backslash I, \sigma_1(i') = \sigma_2(i')$
$\forall i \in I, \sigma_1(i) \neq \sigma_2(i)$
Let's have D and CD such that
$\forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x)$
$(extra(\sigma_1) \neq extra(\sigma_2) \implies I \subset CD)$
Since *up* does not modify the memory, to use the induction hypothesis we only need to prove
$extra(up(\sigma_1)) \neq extra(up(\sigma_2)) \implies I \subset CD$
Which is a consequence of $(extra(\sigma_1) \neq extra(\sigma_2) \implies I \subset CD)$.
So by induction hypothesis: $\exists D_1', VD_1'$ such that

1. $CD, D, t_1 \Downarrow \{(up(\sigma_1), \sigma_1'), (up(\sigma_2), \sigma_2')\}, D_1', VD_1'$
2. $\forall y \in \Delta(\sigma_1', \sigma_2'), I \subset D_1'(y)$
3. $extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD_1'$

We can then have the 3 points:

1. $CD, D, t \Downarrow \{(\sigma_1, \sigma_1'); (\sigma_2, \sigma_2')\}, D_1', VD_1'$
2. $\forall y \in \Delta(\sigma_1', \sigma_2'), I \subset D_1'(y)$
3. $extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD_1'$

- R2:

$$R_2 \frac{up(\sigma_1), t_1 \to \sigma_1'' \qquad next(\sigma_1, \sigma_1''), t_2 \to \sigma_1'}{\sigma_1, t \to \sigma_1'}$$

and,

$$R_2 \frac{up(\sigma_2), t_1 \to \sigma_2'' \qquad next(\sigma_2, \sigma_2''), t_2 \to \sigma_2'}{\sigma_2, t \to \sigma_2'}$$

Let suppose
$\forall i' \in Inputs \backslash \{i\}, \sigma_1(i') = \sigma_2(i')$
$\forall i \in I, \sigma_1(i) \neq \sigma_2(i)$
Let's have D and CD such that
$\forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x)$
$(extra(\sigma_1) \neq extra(\sigma_2) \implies I \subset CD)$
Since *up* does not change the memory but only the extra and $extra(up(\sigma_1)) \neq extra(up(\sigma_2)) \implies I \subset CD$, we can use the induction hypothesis: $\exists D_1', VD_1'$ such that

1. $CD, D, t_1 \Downarrow \{(up(\sigma_1), \sigma_1''), (up(\sigma_2), \sigma_2'')\}, D_1', VD_1'$
2. $\forall y \in \Delta(\sigma_1'', \sigma_2''), I \subset D_1'(y)$
3. $extra(\sigma_1'') \neq extra(\sigma_2'') \implies I \subset VD_1'$

Since the result of *next* has the same memory as the second argument (but not necessarily the same extra), we have

$$\Delta(next(\sigma_1, \sigma_1''), next(\sigma_2, \sigma_2'')) = \Delta(\sigma_1'', \sigma_2'').$$

To use again the induction hypothesis on the second premise, we only need to prove
$extra(next(\sigma_1, \sigma_1'')) \neq extra(next(\sigma_2, \sigma_2'')) \implies I \subset CD'$
Where

$$CD' = CD \cup \begin{cases} VD_1' & \text{if } prod\_extra, \\ \emptyset & \text{otherwise.} \end{cases}$$

If $extra(next(\sigma_1, \sigma_1'')) \neq extra(next(\sigma_2, \sigma_2''))$ then either $extra(\sigma_1) \neq extra(\sigma_2)$ and then by hypothesis $I \subset CD$, or $extra(\sigma_1'') \neq extra(\sigma_2'')$ and then t_1 produces an extra and by hypothesis $I \subset VD_1'$. In both case, $I \subset CD'$
We can then use the induction hypothesis another time: $\exists D_2', VD_2'$ such that

1. $CD', D_1, t_2 \Downarrow \{(next(\sigma_1, \sigma_1''), \sigma_1'), (next(\sigma_2, \sigma_2''), \sigma_2')\}, D_2', VD_2'$
2. $\forall y \in \Delta(\sigma_1', \sigma_2'), I \subset D_2'(y)$
3. $extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD_2'$

We finally have:

1. $CD, D, t \Downarrow \{(\sigma_1, \sigma_1'), (\sigma_2, \sigma_2')\}, D_2', VD_2'$
2. $\forall y \in \Delta(\sigma_1', \sigma_2'), I \subset D_2'(y)$
3. $extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD_2'$

□

B OTHER LEMMAS

The following lemma states that if we have a Pretty-Big-Step derivation, then we can build an annotated multiderivation from it.

Lemma B.1. $\forall \sigma, \sigma', t,$
$\sigma, t \to \sigma' \Rightarrow \quad \forall CD, D,$
$\exists D', VD',$
$CD, D, t \Downarrow \{(\sigma, \sigma')\}, D', VD'$

Proof. Straightforward by induction since the condition needed by every pair of states related by a μ in a multisemantics rule is exactly the condition verified by the pair of state in the corresponding Pretty-Big-Step rule.. □

Lemma B.2 states that if before a multiexecution the context depends on inputs I then the calculated value will also depend on I; and moreover, if a variable or an output xo also depends on I then xo will depend on I at the end of the execution.

Lemma B.2. $\forall CD, D, t, \mu, D', VD'$
$CD, D, t \Downarrow \mu, D', VD'$
$\Rightarrow \quad \forall I, xo,$
$I \subset CD$
$\Rightarrow \quad I \subset VD'$
$\wedge \quad (I \subset D(xo) \Rightarrow I \subset D'(xo))$

Proof. Let have CD, D, t, μ, D', VD' such that we have the multiderivation $CD, D, t \Downarrow \mu, D', VD'$ and prove the lemma by induction on this derivation.

Axiom

$$Ax(t) \frac{\mu = ax_{Some}(\mathtt{fst}(\mu)) \qquad \mu \neq \emptyset}{CD, D, t \Downarrow \mu, D', VD'}$$

With

$$VD' = CD \cup InputRead \bigcup_{x \in VarRead} D(x)$$

$$\forall x, D'(x) = \begin{cases} VD' & \text{if } x \in VarWrite \\ D(x) & \text{otherwise} \end{cases}$$

$$\forall o, D'(o) = \begin{cases} VD' \cup D(o) & \text{if } o \in OutputWrite \\ D(o) & \text{otherwise} \end{cases}$$

Let's have xo a variable or an output and $I \subset CD$. We directly have $I \subset CD \subset VD'$.
Moreover if $I \subset D(xo)$, whether $xo \in VarWrite$, $xo \notin VarWrite$, $xo \in OutputWrite$ or $xo \notin OutputWrite$, we have $I \subset D'(xo)$.

Rule 1

$$R_1(t)\frac{CD, D, t_1 \Downarrow \mu_1, D', VD' \quad \mu = up_{Some}(\texttt{fst}(\mu)) \circ \mu_1}{CD, D, t \Downarrow \mu, D', VD'}$$

Let's have xo a variable or an output and $I \subset CD$.
By induction hypothesis we directly have the result:
$I \subset VD' \quad \wedge \quad (I \subset D(xo) \implies I \subset D'(xo))$

Rule 2

$$R_2(t)\frac{\begin{array}{c} CD, D, t_1 \Downarrow \mu_1, D_1, VD_1 \\ \mu_n = \overrightarrow{up_{Some}}(\texttt{fst}(\mu)) \circ \mu_1 \\ \mu = \mu_n \circ next_{Some}(\texttt{snd}(\mu_n)) \circ \mu_2 \\ CD', D_1, t_2 \Downarrow \mu_2, D', VD' \end{array}}{CD, D, t \Downarrow \mu, D', VD'}$$

With
$$CD' = CD \cup \begin{cases} VD_1 & \text{if } prod\_extra \\ \emptyset & \text{otherwise} \end{cases}$$
Let's have xo a variable or an output and $I \subset CD$.
By induction hypothesis on the first premise we have:
$I \subset VD_1 \quad \wedge \quad (I \subset D(xo) \implies I \subset D_1(xo))$
and by induction on the second:
$I \subset VD' \quad \wedge \quad (I \subset D_1(xo) \implies I \subset D'(xo))$
By combining both implication:
$I \subset VD' \quad \wedge \quad (I \subset D(xo) \implies I \subset D'(xo))$
which is the result we wanted.

Merge

$$Merge(t)\frac{CD, D, t \Downarrow \mu_1, D_1, VD_1 \quad CD, D, t \Downarrow \mu_2, D_2, VD_2 \quad \mu = \mu_1 \cup \mu_2}{CD, D, t \Downarrow \mu, D', VD'}$$

With
$VD' = VD_1 \cup VD_2$
$\forall xo \in Var \cup Outputs, D'(xo) = D_1(xo) \cup D_2(xo)$
Let's have xo a variable or an output and $I \subset CD$.
By induction hypothesis on both premises we have
$I \subset VD_1 \quad \wedge \quad (I \subset D(xo) \implies I \subset D_1(xo))$
and
$I \subset VD_2 \quad \wedge \quad (I \subset D(xo) \implies I \subset D_2(xo))$
Thus,
$I \subset VD' \quad \wedge \quad (I \subset D(xo) \implies I \subset D'(xo))$

We have the result by induction. □

This lemma states that if a variable or an output xo is modified during an execution, then xo depends at least on the context of the execution.

Lemma B.3. $\forall CD, D, t, \mu, D', VD'$
$CD, D, t \Downarrow \mu, D', VD'$
$\Rightarrow \quad \forall \sigma, \sigma', xo,$
$(\sigma, \sigma') \in \mu$
$\Rightarrow \quad \sigma(xo) \neq \sigma'(xo)$
$\Rightarrow \quad CD \subset D'(xo)$

Proof. Let have CD, D, t, μ, D', VD' such that we have the multi-derivation $CD, D, t \Downarrow \mu, D', VD'$ and prove the lemma by induction on this derivation.

Axiom

$$Ax(t)\frac{\mu = ax_{Some}(\texttt{fst}(\mu)) \quad \mu \neq \emptyset}{CD, D, t \Downarrow \mu, D', VD'}$$

where
$$VD' = CD \cup InputRead \bigcup_{x \in VarRead} D(x)$$
$$\forall x, D'(x) = \begin{cases} VD' & \text{if } x \in VarWrite \\ D(x) & \text{otherwise} \end{cases}$$
$$\forall o, D'(o) = \begin{cases} VD' \cup D(o) & \text{if } o \in OutputWrite \\ D(o) & \text{otherwise} \end{cases}$$
Let's have two states σ and σ' such that $(\sigma, \sigma') \in \mu$ and a variable or an output xo such that $\sigma(xo) \neq \sigma'(xo)$. Since $ax(\sigma) = Some\ \sigma'$, $xo \in VarWrite$ or $xo \in OutputWrite$ and thus $CD \subset VD' \subset D'(xo)$.

Rule 1

$$R_1(t)\frac{CD, D, t_1 \Downarrow \mu_1, D', VD' \quad \mu = up_{Some}(\texttt{fst}(\mu)) \circ \mu_1}{CD, D, t \Downarrow \mu, D', VD'}$$

Let's have two states σ and σ' such that $(\sigma, \sigma') \in \mu$ and a variable or an input xo such that $\sigma(xo) \neq \sigma'(xo)$.
Since up only changes the extra and not the memory, we can use the induction hypothesis to ensure:
$$CD \subset D'(xo)$$

Rule 2

$$R_2(t)\frac{\begin{array}{c} CD, D, t_1 \Downarrow \mu_1, D_1, VD_1 \\ \mu_n = \overrightarrow{up_{Some}}(\texttt{fst}(\mu)) \circ \mu_1 \\ \mu = \mu_n \circ next_{Some}(\texttt{snd}(\mu_n)) \circ \mu_2 \\ CD', D_1, t_2 \Downarrow \mu_2, D', VD' \end{array}}{CD, D, t \Downarrow \mu, D', VD'}$$

With
$$CD' = CD \cup \begin{cases} VD_1 & \text{if } prod\_extra \\ \emptyset & \text{otherwise} \end{cases}$$
Let's have two states σ and σ' such that $(\sigma, \sigma') \in \mu$ and a variable or an input xo such that $\sigma(xo) \neq \sigma'(xo)$. Their exists a state σ'' such that $(up(\sigma), \sigma'') \in \mu_1$ and $(next(\sigma, \sigma''), \sigma') \in \mu_2$.
There are two possibilities:
either $\sigma''(xo) \neq \sigma'(xo)$ and thus by induction $CD \subset D'(xo)$;
or $\sigma''(xo) = \sigma'(xo)$ and then $\sigma(xo) \neq \sigma''(xo)$. By induction on the first premise (because $next$ doesn't change the memory of the second argument) we have $CD \subset D_1(xo)$.
And now thanks to lemma B.2, $CD \subset D'(xo)$.

Merge

$$Merge(t)\frac{CD, D, t \Downarrow \mu_1, D_1, VD_1 \quad CD, D, t \Downarrow \mu_2, D_2, VD_2 \quad \mu = \mu_1 \cup \mu_2}{CD, D, t \Downarrow \mu, D', VD'}$$

With
$VD' = VD_1 \cup VD_2$
$\forall x, D'(x) = D_1(x) \cup D_2(x)$

$\forall o, D'(o) = D_1(o) \cup D_2(o)$
Let's have two states σ and σ' such that $(\sigma, \sigma') \in \mu$ and a variable or an output xo such that $\sigma(xo) \neq \sigma'(xo)$.
For symmetric reason, we can suppose $(\sigma, \sigma') \in \mu_1$ and thus the induction hypothesis ensures $CD \subset D'_1(xo) \subset D'(xo)$.

□